CS1760
Multiprocessor Synchronization
Staff

Maurice Herlihy (instructor)
Daniel Engel (grad TA)
Jonathan Lister (HTA)
Bhrath Kayyer (UTA)
Grading

8 Homeworks (40%)

5 programming assignments (20%)

3 Midterms (40%)
Collaboration

Permitted

talking about the homework problems with other students; using other textbooks; using the Internet.

NOT Permitted

obtaining the answer directly from anyone or anything else in any form.
Yes, you can take this course as a capstone course

Only one project possible (concurrent packet filter)

Requires reading ahead of the course

See web page for details
See Course Web Page for …

- TA Hours
- Piazza
- Other important matters
Moore’s Law

Clock speed flattening sharply

Transistor count still rising
Moore’s Law (in practice)
Extinct: the Uniprocessor
Extinct: The Shared Memory Multiprocessor (SMP)
The New Boss: The Multicore Processor (CMP)

All on the same chip

Sun T2000 Niagara
From the 2008 press…

…Intel has announced a press conference in San Francisco on November 17th, where it will officially launch the Core i7 Nehalem processor…

…Sun’s next generation Enterprise T5140 and T5240 servers, based on the 3rd Generation UltraSPARC T2 Plus processor, were released two days ago…
Why is Kunle Smiling?

Niagara 1
Why do we care?

• Time no longer cures software bloat
  – The “free ride” is over

• When you double your program’s path length
  – You can’t just wait 6 months
  – Your software must somehow exploit twice as much concurrency
Traditional Scaling Process

Speedup

1.8x

3.6x

7x

User code

Traditional Uniprocessor

Time: Moore’s law
Ideal Scaling Process

Unfortunately, not so simple…
Actual Scaling Process

Parallelization and Synchronization require great care…
Multicore Programming: Course Overview

- Fundamentals
  - Models, algorithms, impossibility
- Real-World programming
  - Architectures
  - Techniques
Sequential Computation

memory

object

thread

object
Concurrent Computation
Asynchrony

• Sudden unpredictable delays
  – Cache misses (short)
  – Page faults (long)
  – Scheduling quantum used up (really long)
Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays
Road Map

• We are going to focus on principles first, then practice
  – Start with idealized models
  – Look at simplistic problems
  – Emphasize correctness over pragmatism
  – “Correctness may be theoretical, but incorrectness has practical impact”
Concurrency Jargon

- **Hardware**
  - Processors

- **Software**
  - Threads, processes

- Sometimes OK to confuse them, sometimes not.
Parallel Primality Testing

• Challenge
  – Print primes from 1 to $10^{10}$

• Given
  – Ten-processor multiprocessor
  – One thread per processor

• Goal
  – Get ten-fold speedup (or close)
Load Balancing

- Split the work evenly
- Each thread tests range of $10^9$
Procedure for Thread $i$

```java
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10^9+1, j<(i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
```
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
• Need *dynamic* load balancing
Shared Counter

each thread takes a number
int counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
Procedure for Thread $i$

```java
Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```

Shared counter object
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (; j = i*10+1, j<i+1*10; j++) {
    if (isPrime(j))
      print(j);
  }
}
Procedure for Thread $i$

Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}

Stop when every value taken
Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^10) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}

Counter Implementation
Counter Implementation

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```

OK for single thread, not for concurrent threads
What It Means

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```
What It Means

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = temp + 1;
        return temp;
    }
}
```
Not so good...

Value...

1

2

3

2

read 1
write 2
read 2
write 3
read 1
write 2

time

Art of Multiprocessor Programming
Is this problem inherent?

If we could only glue reads and writes together…
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}

Challenge
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}

Make these steps *atomic* (indivisible)
Hardware Solution

public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp  = value;
        value = temp + 1;
        return temp;
    }
}

ReadModifyWrite() instruction
An Aside: Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            long temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```
An Aside: Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            long temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Synchronized block
An Aside: Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Mutual Exclusion
Mutual Exclusion, or “Alice & Bob share a pond”
Alice has a pet
Bob has a pet
The Problem

The pets don’t get along
Formalizing the Problem

• Two types of formal properties in asynchronous computation:
  • Safety Properties
    – Nothing bad happens ever
  • Liveness Properties
    – Something good happens eventually
Formalizing our Problem

• Mutual Exclusion
  – Both pets never in pond simultaneously
  – This is a safety property

• No Deadlock
  – if only one wants in, it gets in
  – if both want in, one gets in.
  – This is a liveness property
Simple Protocol

• Idea
  – Just look at the pond

• Gotcha
  – Not atomic
  – Trees obscure the view
Interpretation

• Threads can’t “see” what other threads are doing
• Explicit communication required for coordination
Cell Phone Protocol

• Idea
  – Bob calls Alice (or vice-versa)

• Gotcha
  – Bob takes shower
  – Alice recharges battery
  – Bob out shopping for pet food …
Interpretation

- Message-passing doesn’t work
- Recipient might not be
  - Listening
  - There at all
- Communication must be
  - Persistent (like writing)
  - Not transient (like speaking)
Can Protocol

cola
cola
Bob conveys a bit
Bob conveys a bit
Can Protocol

• Idea
  – Cans on Alice’s windowsill
  – Strings lead to Bob’s house
  – Bob pulls strings, knocks over cans

• Gotcha
  – Cans cannot be reused
  – Bob runs out of cans
Interpretation

- Cannot solve mutual exclusion with interrupts
  - Sender sets fixed bit in receiver’s space
  - Receiver resets bit when ready
  - Requires unbounded number of interrupt bits
Flag Protocol

Art of Multiprocessor Programming
Alice’s Protocol (sort of)
Bob’s Protocol (sort of)
Alice’s Protocol

• Raise flag
• Wait until Bob’s flag is down
• Unleash pet
• Lower flag when pet returns
Bob’s Protocol

• Raise flag
• Wait until Alice’s flag is down
• Unleash pet
• Lower flag when pet returns

danger!
Bob’s Protocol (2nd try)

- Raise flag
- While Alice’s flag is up
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns
Bob’s Protocol

- Raise flag
- While Alice’s flag is up
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob defers to Alice
The Flag Principle

• Raise the flag
• Look at other’s flag
• Flag Principle:
  – If each raises and looks, then
  – Last to look must see both flags up
Proof of Mutual Exclusion

• Assume both pets in pond
  – Derive a contradiction
  – By reasoning backwards
• Consider the last time Alice and Bob each looked before letting the pets in
• Without loss of generality assume Alice was the last to look…
Proof

Bob last raised flag

Alice last raised her flag

Bob’s last look

Alice’s last look

Alice must have seen Bob’s Flag. A Contradiction
Proof of No Deadlock

• If only one pet wants in, it gets in.
Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.
Proof of No Deadlock

• If only one pet wants in, it gets in.
• Deadlock requires both continually trying to get in.
• If Bob sees Alice’s flag, he backs off, gives her priority (Alice’s lexicographic privilege)
Remarks

• Protocol is *unfair*
  – Bob’s pet might never get in

• Protocol uses *waiting*
  – If Bob is eaten by his pet, Alice’s pet might never get in
Moral of Story

- **Mutual Exclusion cannot be solved by**
  - transient communication (cell phones)
  - interrupts (cans)
- **It can be solved by**
  - one-bit shared variables
  - that can be read or written
The Arbiter Problem (an aside)

Pick a point

Pick a point
The Fable Continues

• Alice and Bob fall in love & marry
The Fable Continues

• Alice and Bob fall in love & marry
• Then they fall out of love & divorce
  – After a coin flip, she gets the pets
  – He has to feed them
The Fable Continues

• Alice and Bob fall in love & marry
• Then they fall out of love & divorce
  – She gets the pets
  – He has to feed them
• Leading to a new coordination problem: Producer-Consumer
Bob Puts Food in the Pond
Alice releases her pets to Feed
Producer/Consumer

• Alice and Bob can’t meet
  – Each has restraining order on other
  – So he puts food in the pond
  – And later, she releases the pets

• Avoid
  – Releasing pets when there’s no food
  – Putting out food if uneaten food remains
Producer/Consumer

- Need a mechanism so that
  - Bob lets Alice know when food has been put out
  - Alice lets Bob know when to put out more food
Surprise Solution
Bob puts food in Pond
Bob knocks over Can
Alice Releases Pets

Art of Multiprocessor Programming
Alice Resets Can when Pets are Fed
while (true) {
    while (can.isUp()){}
    pet.release();
    pet.recapture();
    can.reset();
}
while (true) {
    while (can.isUp()){};
    pet.release();
    pet.recapture();
    can.reset();
}

Alice’s code

while (true) {
    while (can.isDown()){};
    pond.stockWithFood();
    can.knockOver();
}

Bob’s code
Correctness

• Mutual Exclusion
  – Pets and Bob never together in pond
Correctness

• Mutual Exclusion
  – Pets and Bob never together in pond

• No Starvation
  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
Correctness

- **Mutual Exclusion**
  - Pets and Bob never together in pond

- **No Starvation**
  - If Bob always willing to feed, and pets always famished, then pets eat infinitely often.

- **Producer/Consumer**
  - The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.
Could Also Solve Using Flags
Waiting

• Both solutions use waiting
  – \texttt{while(mumble)}{}

• In some cases waiting is \textit{problematic}
  – If one participant is delayed
  – So is everyone else
  – But delays are common & unpredictable
The Fable drags on …

• Bob and Alice still have issues
The Fable drags on …

- Bob and Alice still have issues
- So they need to communicate
The Fable drags on …

- Bob and Alice still have issues
- So they need to communicate
- They agree to use billboards …
Billboards are Large

Letter Tiles
From Scrabble™ box

Art of Multiprocessor Programming
Write One Letter at a Time …
To post a message
Let’s send another message
Uh-Oh
Readers/Writers

• Devise a protocol so that
  – Writer writes one letter at a time
  – Reader reads one letter at a time
  – Reader sees “snapshot”
    • Old message or new message
    • No mixed messages
Readers/Writers (continued)

• Easy with mutual exclusion

• But mutual exclusion requires waiting
  – One waits for the other
  – Everyone executes sequentially

• Remarkably
  – We can solve R/W without mutual exclusion
Esoteric?

• Java container `size()` method
• Single shared counter?
  – incremented with each `add()` and
  – decremented with each `remove()`
• Threads wait to exclusively access counter

performance bottleneck
Readers/Writers Solution

• Each thread $i$ has $\text{size}[i]$ counter
  – only it increments or decrements.
• To get object’s size, a thread reads a “snapshot” of all counters
• This eliminates the bottleneck
Why do we care?

- We want as much of the code as possible to execute concurrently (in parallel)
- A larger sequential part implies reduced performance
- Amdahl’s law: this relation is not linear…
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{\frac{1}{\text{1-thread execution time}} + \frac{n}{\text{n-thread execution time}}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

Parallel fraction
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

Sequential fraction

Parallel fraction
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]

Sequential fraction

Parallel fraction

Number of threads

1

\(p\)

\(n\)
Bad synchronization ruins everything
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}
\]
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}
\]
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.9 + \frac{0.9}{10}} = 5.26
\]
Example

• Ten processors
• 99% concurrent, 01% sequential
• How close to 10-fold speedup?
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]
Back to Real-World Multicore Scaling

1.8x  2x  2.9x

Speedup

User code

Multicore

Not reducing sequential % of code

Art of Multiprocessor Programming
Shared Data Structures

Coarse Grained

Fine Grained

25% Shared

75% Unshared

25% Shared

75% Unshared
Shared Data Structures

Why only 2.9 speedup

Coarse Grained

Fine Grained

25% Shared

75% Unshared

25% Shared

75% Unshared

Honk!
Shared Data Structures

Why fine-grained parallelism matters

Coarse Grained

- 25% Shared
- 75% Unshared

Fine Grained

- 25% Shared
- 75% Unshared

---

Honk!

Honk!

Honk!
This course is about the parts that are hard to make concurrent ... but still have a big influence on speedup!